Design of Practical Succinct Data Structures for Large Data Collections

Roberto Grossi and Giuseppe Ottaviano

Dipartimento di Informatica
Università di Pisa
{grossi,ottaviano}@di.unipi.it

Abstract. We describe a set of basic succinct data structures which have been implemented as part of the *Succinct* library, and applications on top of the library: an index to speed-up the access to collections of semi-structured data, a compressed string dictionary, and a compressed dictionary for scored strings which supports top-k prefix matching.

1 Introduction

Succinct data structures (SDS) encode data in small space and support efficient operations on them. Encoding is a well studied problem in information theory and there is a simple lower bound on the required space in bits: if data are entries from a domain D, encoding each entry with less than $\lceil \log |D| \rceil$ bits cannot uniquely identify all the entries in D (here logs are to the base 2). Thus any correct encoding requires at least $\lceil \log |D| \rceil$ bits in the worst case, which is known as the *information-theoretic lower bound*. Variants of this concept use some form of entropy of D or other adaptive measures for the space occupancy, in place of the $\lceil \log |D| \rceil$ term, but the idea is essentially the same.

Going one step beyond encoding data, SDS can also retrieve data in response to queries. When data are any given subset of elements, an example of query is asking if an input element belongs to that subset. Without any restriction on the execution time of the queries, answering them becomes a trivial task to perform: simply decode the whole encoded data and scan to answer the queries. The challenge in SDS is to quickly perform its query operations, possibly in *constant time* per query. To attain this goal, SDS can use extra r bits of *redundancy* in addition to those indicated by the information-theoretic lower bound.

SDS have been mainly conceived in a theoretical setting. The first results date back to Elias' papers of 1974 and 1975 on information retrieval [9,10] with a reference to the Minsky-Papert problem on searching with bounded Hamming distance. The power of SDS has been extensively discussed in Jacobson's PhD thesis [17], where he shows how to store data such as binary sequences, trees, and graphs in small space. The design of SDS is also linked to the bit probe complexity of data structures, see the literature cited in [22,6], and the time-space tradeoffs of data structures, e.g. [27]. As of now, there are SDS for sets of integers, sequences, strings, trees, graphs, relations, functions, permutations, geometric data, special formats (XML, JSON), and so on [19].

What is interesting for the algorithm engineering community is that, after some software prototype attempts, efficient libraries for SDS are emerging, such as the C++ libraries *libcds* [21], *rsdic* [29], *SDSL* [31], and the Java/C++ library *Sux* [33]. They combine the advantage of data compression with the performance of efficient data structures, so that they can operate directly on compressed data by accessing just a *small* portion of data that needs to be decompressed. The field of applications is vast and the benefit is significant for large data sets that can be kept in main memory when encoded in succinct format.

The preprocessing stage of SDS builds an *index* that occupies r bits (of redundancy). Systematic SDS have a clear separation of the index from the compressed data, with several advantages [3]. When this separation is not obtained, the resulting SDS are called non-systematic because many bits contribute simultaneously both to the index and the compressed data.[1]

Given data chosen from a domain D, the designer of SDS aims at using $r + \lceil \log |D| \rceil$ bits to store data+index, with the main goal of asymptotically minimizing both the space redundancy r and the query time, or at least finding a good trade-off for these two quantities. Optimality is achieved when r and query bounds match the corresponding lower bounds. The problem is not only challenging from a theoretical point of view, where sophisticated upper and lower bounds have been proposed, e.g. [26,35]. Its practical aspects are quite relevant, e.g. in compressed text indexing [12,15], where asymptotically small redundancy and query time quite often do not translate into practical and fast algorithms.

We focus on succinct indexes for semi-structured data and strings but we believe that many SDS should be part of the modern algorithmist's toolbox for all the several data types mentioned so far. We survey a set of fundamental SDS and primitives to represent and index ordered subsets, sequences of bits, sequences of integers, and trees, that can be used as building blocks to build more sophisticated data structures. These SDS have proven to be practical and mature enough to be used as black boxes, without having to understand the inner details. The SDS that we describe are all implemented as part of the *Succinct* C++ library [32]; we give three examples of its applications in the last section.

2 Basic Toolkit for the Designer

In this section we define the most common primitives used when designing SDS, which are also sufficient for the applications of Sect. 4. The details on the algorithm and data structures used to implement them, along with their time and space complexities, are deferred to Sect. 3.

2.1 Subsets and Bitvectors

Bitvectors are the basic building block of SDS. In fact, most constructions consist of a collection of bitvectors and other sequences, that are aligned under some

[1] In a certain sense, implicit data structures as the binary heap can be seen as a form of non-systematic SDS where $r = O(\log n)$ bits for n elements.

logic that allows to efficiently translate a position in one sequence to a position in another sequence; this pattern will be used extensively in Sect. 4.

Formally, a *bitvector* is a finite sequence of values from the alphabet $\{\mathtt{0}, \mathtt{1}\}$, i.e. *bits*. For a bitvector s we use $|s|$ to denote its length, and s_i to denote its i-th element, where i is 0-based. Bitvectors can be interpreted as *subsets* of an ordered universe U: the ordering induces a numbering from 0 to $|U| - 1$, so a subset X can be encoded as a bitvector s of length $|U|$, where s_i is `1` if the i-th element of U belongs to X, and `0` otherwise. Here the information-theoretic lower bound is $\lceil \log \binom{|U|}{m} \rceil$ where m is the number of `1`s [28].

Rank and Select. The Rank and Select primitives form the cornerstone of most SDS, since they are most prominently used to align different sequences under the pattern described above. These operations can be defined on sequences drawn from arbitrary alphabets, but for simplicity we will focus on bitvectors.

- $\mathrm{Rank}_\mathtt{1}(i)$ returns the number of occurrences of `1` in $s[0, i)$.
- $\mathrm{Select}_\mathtt{1}(i)$ returns the position of the i-th occurrence of `1`.

An operation tightly related to Rank/Select is $\mathrm{Predecessor}_\mathtt{1}(i)$, which returns the position of the *rightmost* occurrence of `1` preceding or equal to i. Note that $\mathrm{Rank}_\mathtt{1}(\mathrm{Select}_\mathtt{1}(i)) = i$ and $\mathrm{Select}_\mathtt{1}(\mathrm{Rank}_\mathtt{1}(i)) = \mathrm{Predecessor}_\mathtt{1}(i)$ (Symmetrically, $\mathrm{Rank}_\mathtt{0}$, $\mathrm{Select}_\mathtt{0}$, and $\mathrm{Predecessor}_\mathtt{0}$ can be defined on occurrences of `0`s.)

If the bitvector is interpreted as a subset under the correspondence defined above, $\mathrm{Rank}_\mathtt{1}$ returns the number of elements in the subset that are strictly smaller than a given element of the universe, while $\mathrm{Select}_\mathtt{1}$ returns the elements of the subset in sorted order. $\mathrm{Predecessor}_\mathtt{1}$ returns the largest element that is smaller or equal to a given element of the universe.

A basic example of how to use $\mathrm{Rank}_\mathtt{1}$ to align different sequences is the *sparse array*. Let A be an array of n elements of k bits each; then, to store it explicitly, kn bits are needed. However, if a significant number of elements is 0 (or any fixed constant), we can use a bitvector z of n bits to encode which elements are zero and which ones are not, and store only the non-zeros in an another array B. To retrieve $A[i]$, we return 0 if $z_i = 0$; otherwise, the value is $B[\mathrm{Rank}_\mathtt{1}(i)]$. This allows to reduce the space from kn bits to $n + k|B|$ bits, plus the space taken by the data structure used to efficiently support $\mathrm{Rank}_\mathtt{1}$.

2.2 Balanced Parentheses and Trees

Another class of sequences of particular interest is given by sequences of *balanced parentheses* (BP), which can be used to represent arbitrary trees in spaces close to the information-theoretic optimum.

BP sequences are inductively defined as follows: an empty sequence is BP; if α and β are sequences of BP, then also `(`α`)`β is a sequence of BP, where `(` and `)` are called *mates*. For example, $s =$ `(()(()()))` is a sequence of BP. These sequences are usually represented as bitvectors, where `1` represents `(` and `0` represents `)`.

Several operations operations can be defined on such sequences; as we will see shortly, the following ones are sufficient for basic tree navigation.

- FindClose(i), for a value i such that $s_i =$ `(`, returns the position $j > i$ such that $s_j =$ `)` is its mate. (FindOpen(i) is defined analogously.)
- Enclose(i), for a value i such that $s_i =$ `(`, returns the position $j < i$ such that $s_j =$ `(` and the pair of j and its mate enclose the pair of i and its mate.
- $\text{Rank}_{(}(i)$ returns the pre-order index of the node corresponding to the parenthesis at position i and its mate; this is just the number of open parentheses preceding i.
- Excess(i) returns the difference between the number of open parentheses and that of close parentheses in the first $i + 1$ positions of s. The sequence of parentheses is balanced if and only if this value is always non-negative, and it is easy to show that it equals $2 \cdot \text{Rank}_{(}(i) - i$.

Balanced Parentheses Tree Encoding (BP). The BP representation of trees was introduced by Munro and Raman [23]. A sequence of BP implicitly represents an ordinal tree, where each node corresponds to a pair of mates. By identifying each node with the position p of its corresponding open parenthesis, several traversal operations can be reduced to the operations defined above.

Depth-First Unary Degree Sequence (DFUDS). Another tree representation based on balanced parentheses was introduced by Benoit et al. [5]. Called *depth-first unary degree sequence* (DFUDS), it is constructed by concatenating in depth-first order the node degrees encoded in unary, i.e. a degree d is encoded as $\texttt{(}^d\texttt{)}$. It can be shown that by prepending an initial `(`, the obtained sequence of parentheses is balanced.

By identifying each node of the tree with the position p of beginning of its degree encoding, traversal operations can be mapped to sequence operations. Compared to the BP representation we loose the Depth operation, but we gain the operation Child which returns the i-th child by performing a single FindClose.

Figure 1 shows an example of a tree represented with BP and DFUDS encodings. Note that both encodings require just 2 bits per node, plus the data structures needed to support the operations. It can be shown that the information-theoretic lower bound to represent an arbitrary tree of n nodes is $2n - o(n)$ bits, so both encodings are asymptotically close to the lower bound.

2.3 Range Minimum Queries

Given a sequence A of elements drawn from a totally ordered universe, a *Range Minimum Query* for the range $[i, j]$, denoted as $\text{RMQ}(i, j)$, returns the position of the minimum element in $A[i, j]$ (returning the leftmost in case of ties).

This operation finds application, for example, in suffix arrays, to find the LCP (Longest Common Prefix) of a range of suffixes by using the vector of LCPs of consecutive suffixes: the LCP of the range is just the minimum among the consecutive LCPs. Another application is top-k retrieval (see Sect. 4).

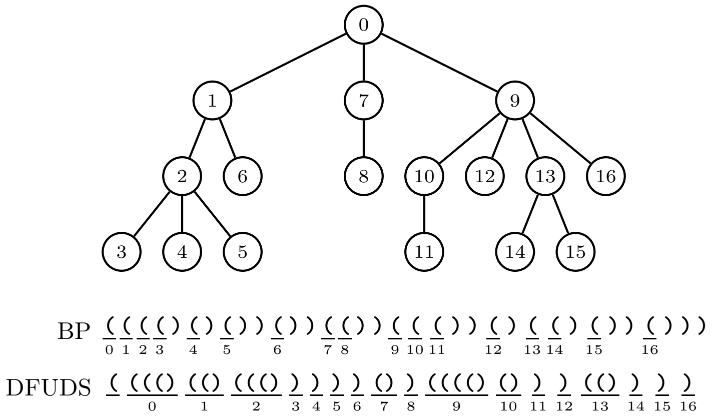


Fig. 1. BP and DFUDS encodings of an ordinal tree

3 Toolkit Implementation

For the operations described in Sect. 2 there exist SDS that can support them in constant time while taking only $r = o(n)$ bits of redundancy, meaning that as the input size n grows to infinity the relative overhead per element of the sequence goes to zero. This fact is the raison d'être for the whole field of SDS: it means that, in a reasonably realistic model of computation, using SDS instead of classical pointer-based ones involves no runtime overhead, and the space overhead needed to support the primitives is negligible.

In practice, however, the constants hidden in the $o(\cdot)$ and $O(\cdot)$ notations are large enough that these data structures become competitive with the classical ones only at unrealistic data sizes. Furthermore, CPU cache and instruction-level parallelism play an important role in the performance of these algorithms, but it is impossible to appreciate their influence in an abstract model of computation.

In recent years, a large effort has been devoted to the *algorithm engineering* of SDS, producing practical data structures that are fast and space-efficient even for small inputs. When compared to their theoretical counterparts, the time bounds often grow from $O(1)$ to $O(\log n)$, and the space bounds from $o(n)$ to $O(n)$, but for all realistic data sizes they are more efficient in both time and space.

To give a sense of how these data structures work, we give a detailed explanation of how Rank can be implemented. The other data structures are only briefly summarized and we refer to the relevant papers for a complete description.

All the SDS described here are *static*, meaning that their contents cannot be changed after the construction. While there has been a significant amount of work in *dynamic* SDS, most results are theoretical, and the practice has not caught up yet. The engineering of dynamic SDS is certainly an interesting research topic, which we believe will receive significant attention in the next few years.

3.1 The *Succinct* Library

The SDS structures described in this section, which to date are among the most efficient, are implemented as part of the *Succinct* library [32]. The library is available with a permissive license, in the hope that it will be useful both in research and applications. While similar in functionality to other existing C++ libraries such as libcds [21], SDSL [31], and Sux [33], we made some radically different architectural choices, which we describe below.

Memory Mapping. As in most static data structures libraries, all the data structures in *Succinct* can be serialized to disk. However, as opposed to libcds, SDSL, and Sux, deserialization is performed by *memory mapping* the data structure, rather than *loading* it into memory.

While being slightly less flexible, memory mapping has several advantages over loading. For example, for short-running processes it is often not necessary to load the full data structure in memory; instead the kernel will load only the relevant pages. If such pages were accessed recently, they are likely to be still in the kernel's page cache, thus making the startup even faster. If several processes access the same structure, the memory pages that hold it are shared among all the processes; with loading, instead, each process keeps its own private copy of the data. Lastly, if the system runs out of memory, it can just un-map unused pages; with loading, it has to *swap* them to disk, thus increasing the I/O pressure.

For convenience we implemented a mini-framework for serialization/memory mapping which uses template metaprogramming to describe recursively a data structure through a single function that lists the members of a class. The mini-framework then automatically implements serialization and mapping functions.

Templates Over Polymorphism. We chose to avoid dynamic polymorphism and make extensive use of C++ templates instead. This allowed us to write idiomatic and modular C++ code without the overhead of virtual functions.

Multi-platform 64-Bit Support. The library is tested under Linux, Mac OS X, and Microsoft Windows, compiled with gcc, clang and MSVC. Like Sux and parts of SDSL, *Succinct* is designed to take advantage of 64-bit architectures, which allow us to use efficient broadword algorithms [20] to speed up several operations on memory words. Another advantage is that the data structures are not limited to 2^{32} elements or less like 32-bit based implementations, a crucial requirement for large datasets, which are the ones that benefit the most from SDS. We also make use of CPU instructions that are not exposed to C++ but are widely available, such as those to retrieve the MSB and LSB of a word, or to reverse its bytes. While all these operations can all be implemented with broadword algorithms, the specialized instructions are faster.

3.2 Rank/Select

Like many SDS, Jacobson's implementation [17] of Rank relies on Four Russians trick by splitting the data into pieces that are small enough that all the answers to the queries for the small pieces can be *tabulated* in small space, and answers to queries on the whole input can be assembled from queries on the small pieces and on a sparse global data structure. Specifically, the input is divided into *super-blocks* of size $\log^2 n$, and the answer to the Rank at the beginning of each super-block is stored in an array. This takes $O(\#\text{blocks} \cdot \log n) = O(n \log n / \log^2 n) = o(n)$.

The super-blocks are then divided into *blocks* of size $\frac{1}{2} \log n$, and the answers to the Rank queries at the beginning of each block are stored *relative to their super-block*: since the superblock is only $\log^2 n$ bits long, the relative ranks cost $O(\log \log n)$ bits each, so overall the block ranks take $O(n \log \log n / \log n) = o(n)$.

The blocks are now small enough that we can *tabulate* all the possible Rank(i) queries: the number of different blocks is at most $2^{\frac{\log n}{2}} = O(\sqrt{n})$, the positions in the block are at most $O(\log n)$, and each answer requires $O(\log \log n)$ bits, so overall the space needed by the table is $O(\sqrt{n} \log n \log \log n) = o(n)$.

To answer Rank(i) on the whole bitvector it is sufficient to sum the rank of its super-block, the relative rank of its block, and the rank inside the block. All three operations take $O(1)$ time, so overall the operation takes constant time.

This data structure can be implemented almost as described, but it is convenient to have fixed-size blocks and super-blocks. This yields an $O(n)$-space data structure, but the time is still $O(1)$. Furthermore, if the blocks are sized as the machine word, the in-block Rank can be efficiently computed with broadword operations, as suggested by Vigna [34], hence avoiding to store the table.

The constant-time data structure for Select is significantly more involved: its practical alternative is to perform a binary search on the super-block ranks, followed by a linear search in the block partial ranks and then inside the blocks. The binary search can be made more efficient by storing the answer to $\text{Select}_{\mathtt{1}}$ for every k-th `1`, so that the binary search can be restricted to a range that contains at most k ones. This algorithm, which is known as *hinted binary search*, can take $O(\log n)$ time but is extremely efficient in practice.

In *Succinct*, Rank and Select are implemented in the `rs_bit_vector` class, which uses the `rank9` data structure [34].

3.3 Elias-Fano Representation of Monotone Sequences

The *Elias-Fano representation of monotone sequences* [9,11] is an encoding scheme to represent a non-decreasing sequence of m integers $\langle x_1, \cdots, x_m \rangle$ from the universe $[0..n)$ occupying $2m + m \left\lceil \log \frac{n}{m} \right\rceil + o(m)$ bits, while supporting constant-time access to the i-th integer. It can be effectively used to represent sparse bitvectors (i.e. where the number m of ones is small with respect to the size n of the bitvector), by encoding the sequence of the positions of the ones. Using this representation the retrieval of the the i-th integer can be interpreted as $\text{Select}_{\mathtt{1}}(i)$, and similarly it is possible to support $\text{Rank}_{\mathtt{1}}$.

The scheme is very simple and elegant. Let $\ell = \lfloor \log(n/m) \rfloor$. Each integer x_i is first encoded in binary into $\lceil \log n \rceil$ bits, and the binary encoding is then split into the first $\lceil \log n \rceil - \ell$ *higher* bits and the last ℓ *lower* bits. The sequence of the higher bits is represented as a bitvector H of $\lceil m + n/2^\ell \rceil$ bits, where for each i, if h_i is the value of the higher bits of x_i, then the position $h_i + i$ of H is set to `1`; H is `0` elsewhere. The lower bits of each x_i are just concatenated into a bitvector L of $m\ell$ bits. To retrieve the i-th integer we need to retrieve its higher and lower bits and concatenate them. The lower bits are easily retrieved from L. To retrieve the upper bits it is sufficient to note that $h_i = \text{Select}_H(\mathtt{1}, i) - i$. The implementation is straightforward, provided that Select is supported on H.

In *Succinct*, we implement it using the `darray` data structure [24], which supports Select in $O(1)$ time without requiring a data structure to support Rank. The class `elias_fano` implements a sparse bitvector encoded with Elias-Fano.

3.4 Balanced Parentheses

To implement operations on balanced parentheses, variants of a data structure called Range-Min-Max tree [30] have proven the most effective in practice [2], despite their $O(\log n)$ time and $O(n)$ space. The data structure divides the sequence into a hierarchy of blocks, and stores the minimum and maximum Excess value for each block. This yields a tree of height $O(\log n)$ height, that can be traversed to find mate and enclosing parentheses.

In *Succinct* the class `bp_vector` implements the basic operations on balanced parentheses. It uses a variant of the Range-Min-Max tree called *Range-Min tree* [14], which only stores the minimum excess, thus halving the space occupancy with respect to the Range-Min-Max tree. This weakens the range of operations that the data structure can support, but all the important tree navigation operations can be implemented.

3.5 Range Minimum Queries

The RMQ problem is intimately related to the *Least Common Ancestor* (LCA) problem on trees. In fact, the LCA problem can be solved by reducing it to an RMQ on a sequence of integers derived from the tree, while RMQ can be solved by reducing it to an LCA in a tree derived from the sequence, called the *Cartesian Tree* [4]. As shown by Fischer and Heun [13], RMQ on an array A can be reduced to an RMQ on the excess sequence of the DFUDS representation of the *2d-Min-Heap*, which, as noted by Davoodi et al. [7], is an alternative representation of the Cartesian tree. The RMQ operation on an excess sequence can be easily implemented by using the Range-Min tree, so since the DFUDS representation is already equipped with a Range-Min tree to support navigational operations, no extra space is needed.

In *Succinct*, RMQ is implemented in the class `cartesian_tree`. The algorithm is a minor variation, described in [16], of the scheme by Fischer and Heun [13].

4 Applications

We describe some applications of the *Succinct* library that involve handling large collections of semi-structured or textual data. Interestingly, SDS are gaining popularity in other communities such as bioinformatics, Web information retrieval, and networking.

4.1 Semi-indexing Semi-structured Data

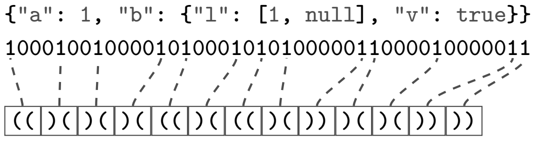
With the advent of large-scale distributed processing systems such as MapReduce [8] and Hadoop [1] it has become increasingly common to store large amounts of data in textual semi-structured formats such as JSON and XML, as opposed to the structured databases of classical data warehousing. The flexibility of such formats allows to evolve the data schema without having to migrate the existing data to the new schema; this is particularly important in logging applications, where the set of attributes to be stored usually evolves quickly.

The disadvantage of such formats is that each record, represented by a semi-structured document, must be parsed completely to extract its attributes; in typical applications, the records contain several attributes but each query requires a different small subset of such attributes, thus reading the full document can be highly inefficient. Alternative representations for semi-structured data have been proposed in the literature, many using SDS, that are both compact and support efficient querying of attributes, but they rely on changing the data format, which may be unacceptable in some scenarios where compatibility or interoperability with existing tools is required.

In [25] it was introduced the concept of *semi-index*, which is a systematic SDS that can be used to speed-up the access to an existing semi-structured document without changing its format: the semi-index is stored on a separate file and it takes only a small fraction of the size of the original data.

The semi-index consists of two components: a *positional index*, that is an index of the positions in the document that are starting points of its elements, and a succinct encoding of the parse tree of the document. The positional index can be represented with a bitvector and encoded with Elias-Fano, while the parse tree can be represented with a BP sequence.

For JSON documents [18], the positional index has a `1` in correspondence of the positions where either of the characters `{}[],:` occur. For each one of these, a pair of parentheses is appended to the BP sequence, specifically `((` for `{` and `[`, `))` for `}` and `]`, and `)(` for `,` and `:`. An example is shown in Fig. 2. It can be shown that the BP sequence is indeed balanced, and that it is possible to navigate the parse tree of the document by accessing the two index sequences and a minimal part of the original document. An experimental analysis on large-scale document collections has shown speed-ups between 2.5 and 10 times compared to parsing each document, while the space overhead given by the semi-index is at most $\sim 10\%$.



```
{"a": 1, "b": {"l": [1, null], "v": true}}
10001001000001010001010100000011000010000011
```

Fig. 2. Semi-index of a JSON document

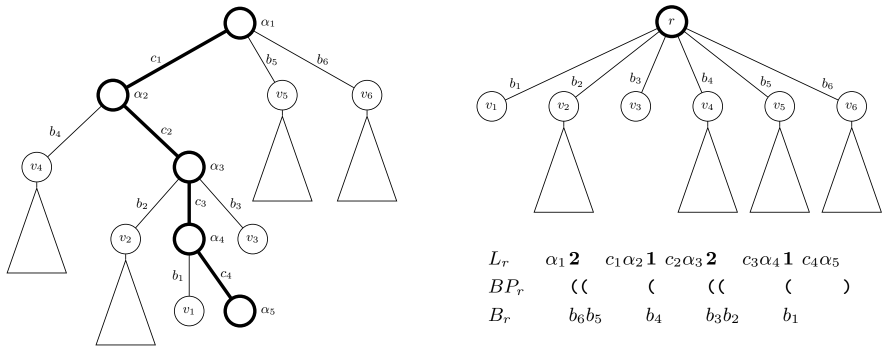
4.2 Compressed String Dictionaries

A string dictionary is a data structure that maps bijectively a set of n strings to the integer range $[0, n)$. String dictionaries are among the most fundamental data structures, and find application in basically every algorithm that handles strings.

In many scenarios where the string sets are large, the strings are *highly redundant*; a *compressed string dictionary* exploits this redundancy to reduce the space usage of the data structure. Traditionally, string dictionaries are implemented by using *tries*, which are not only fast, but they also offer some degree of compression by collapsing the common prefixes. Tries represented with SDS offer even higher space savings, but the performance suffers a large slow-down because tries can be highly unbalanced, and navigational operations in succinct trees can be costly. Furthermore, prefix compression is not effective for strings that might share other substrings besides the prefix.

In [14] it was introduced a succinct representation for tries that makes use of *path decompositions*, a transformation that can turn a unbalanced tree into a balanced one. The representation also enables compression of the trie *labels*, thus exploiting the redundancy of frequent substrings.

A *path decomposition* of a trie $\mathcal{T}$ is a tree $\mathcal{T}^c$ whose nodes correspond to node-to-leaf paths in $\mathcal{T}$. The tree is built by first choosing a root-to-leaf path π in $\mathcal{T}$ and associating it with the root node u_π of $\mathcal{T}^c$; the children of u_π are defined recursively as the path decompositions of the subtries hanging off the path π, and their edges are labeled with the labels of the edges from the path π to the subtries. An example is shown in Fig. 3. The resulting tree is then encoded using a DFUDS representation, and the sequence of labels is compressed with a simple dictionary compression scheme.



α_1
c_1
b_5
b_6
α_2
v_5
v_6
b_4
c_2
v_4
α_3
b_2
c_3
b_3
v_2
α_4
v_3
b_1
c_4
v_1
α_5
r
b_1 b_2 b_3 b_4 b_5 b_6
v_1 v_2 v_3 v_4 v_5 v_6
L_r α_1 **2** $c_1\alpha_2$ **1** $c_2\alpha_3$ **2** $c_3\alpha_4$ **1** $c_4\alpha_5$
BP_r (((((( )
B_r b_6b_5 b_4 b_3b_2 b_1

Fig. 3. Path decomposition of a trie. The α_i denote the labels of the trie nodes, c_i and b_i the branching characters (depending on whether they are on the path or not).

Depending on the strategy used to choose the decomposition path π, different properties can be obtained. If we start from the root and recursively choose the child with most descendents, the resulting path is called a *centroid* path, and the resulting *centroid path decomposition* has height $O(\log n)$, regardless of whether the trie is balanced or not. Alternatively, we can choose recursively the leftmost child; the resulting decomposition is called *lexicographic path decomposition* because the numbers associated to the strings of the set respect the lexicographic ordering, but for this decomposition no height guarantees can be given.

In the experiments, large-scale collections of URLs, queries, and web page titles can be compressed down to 32% to 13% of their original size, while maintaining access and lookup times of few microseconds.

4.3 Top-k Completion in Scored String Sets

Virtually every modern application, either desktop, web, or mobile, features some kind of auto-completion of text-entry fields. Specifically, as the user enters a string one character at a time, the system presents k suggestions to speed up text entry, correct spelling mistakes, and help users formulate their intent.

This can be thought of as having a *scored* string set, meaning that each string is assigned a score, and given a prefix p we want to find the k strings prefixed by p that have highest score. We call this problem *top-k completion.* Since the sets of suggestion strings are usually large, space-efficiency is crucial.

A simple solution is to combine the lexicographic trie of Sect. 4.2 with an RMQ data structure. The strings in the string set are stored in the trie, and their scores are stored in lexicographic order in an array R. The set of the indexes of strings that start with a given prefix p is a contiguous range $[a, b]$.

We can then compute $r = \mathrm{RMQ}(a, b)$ to find the index r of the highest-scored string prefixed by p. The second string is either the highest-scored one in $[a, r-1]$ or in $[r+1, b]$, and so on. By using a priority queue it is possible to retrieve the top-k completions one at a time.

It is however possible to do better: we can use again the tries of Sect. 4.2, but with a different path decomposition, where the chosen decomposition path π is the path from the root to the highest-scored leaf. It can be shown that in the resulting *max-score path decomposition* it is possible to find the top-k completions of a given node by visiting just k nodes; since, as noted before, navigational operations are costly in succinct representations, performance is significantly better than the RMQ-based solution.

On large sets of queries and URLs, experiments have shown compression ratios close or better than those of `gzip`, with average times per completion of about one microsecond [16].

References

1. Apache Hadoop, `http://hadoop.apache.org/`
2. Arroyuelo, D., Cánovas, R., Navarro, G., Sadakane, K.: Succinct trees in practice. In: ALENEX, pp. 84–97 (2010)
3. Barbay, J., He, M., Munro, J.I., Satti, S.R.: Succinct indexes for strings, binary relations and multilabeled trees. ACM Transactions on Algorithms 7(4), 52 (2011)

4. Bender, M.A., Farach-Colton, M.: The LCA problem revisited. In: Gonnet, G.H., Viola, A. (eds.) LATIN 2000. LNCS, vol. 1776, pp. 88–94. Springer, Heidelberg (2000)
5. Benoit, D., Demaine, E.D., Munro, J.I., Raman, R., Raman, V., Rao, S.S.: Representing trees of higher degree. Algorithmica 43(4), 275–292 (2005)
6. Buhrman, H., Miltersen, P.B., Radhakrishnan, J., Venkatesh, S.: Are bitvectors optimal? SIAM Journal on Computing 31(6), 1723–1744 (2002)
7. Davoodi, P., Raman, R., Satti, S.R.: Succinct representations of binary trees for range minimum queries. In: Gudmundsson, J., Mestre, J., Viglas, T. (eds.) COCOON 2012. LNCS, vol. 7434, pp. 396–407. Springer, Heidelberg (2012)
8. Dean, J., Ghemawat, S.: MapReduce: Simplified data processing on large clusters. In: OSDI, pp. 137–150 (2004)
9. Elias, P.: Efficient storage and retrieval by content and address of static files. Journal of the ACM (JACM) 21(2), 246–260 (1974)
10. Elias, P., Flower, R.A.: The complexity of some simple retrieval problems. Journal of the ACM 22(3), 367–379 (1975)
11. Fano, R.: On the number of bits required to implement an associative memory. Memorandum 61. Computer Structures Group, Project MAC. MIT (1971)
12. Ferragina, P., Manzini, G.: Indexing compressed text. J. ACM 52(4), 552–581
13. Fischer, J., Heun, V.: Space-efficient preprocessing schemes for range minimum queries on static arrays. SIAM J. Comput. 40(2), 465–492 (2011)
14. Grossi, R., Ottaviano, G.: Fast compressed tries through path decompositions. In: ALENEX, pp. 65–74 (2012)
15. Grossi, R., Vitter, J.S.: Compressed suffix arrays and suffix trees with applications to text indexing and string matching. SIAM J. Comput. 35(2), 378–407 (2005)
16. Hsu, B.J.P., Ottaviano, G.: Space-efficient data structures for top-k completion. In: Proceedings of the 22st World Wide Web Conference (WWW) (2013)
17. Jacobson, G.: Space-efficient static trees and graphs. In: FOCS, pp. 549–554 (1989)
18. JSON specification, `http://json.org/`
19. Kao, M.Y. (ed.): Encyclopedia of Algorithms. Springer (2008)
20. Knuth, D.E.: The Art of Computer Programming. Fascicle 1: Bitwise Tricks & Techniques; Binary Decision Diagrams, vol. 4. Addison-Wesley (2009)
21. libcds - Compact Data Structures Library, `http://libcds.recoded.cl/`
22. Miltersen, P.B.: The bit probe complexity measure revisited. In: Enjalbert, P., Wagner, K.W., Finkel, A. (eds.) STACS 1993. LNCS, vol. 665, pp. 662–671. Springer, Heidelberg (1993)
23. Munro, J.I., Raman, V.: Succinct representation of balanced parentheses and static trees. SIAM Journal on Computing 31(3), 762–776 (2001)
24. Okanohara, D., Sadakane, K.: Practical entropy-compressed rank/select dictionary. In: ALENEX (2007)
25. Ottaviano, G., Grossi, R.: Semi-indexing semi-structured data in tiny space. In: CIKM, pp. 1485–1494 (2011)
26. Pătraşcu, M.: Succincter. In: FOCS 2008, pp. 305–313 (2008)
27. Pătraşcu, M., Thorup, M.: Time-space trade-offs for predecessor search. In: STOC, pp. 232–240 (2006)
28. Raman, R., Raman, V., Satti, S.R.: Succinct indexable dictionaries with applications to encoding k-ary trees, prefix sums and multisets. ACM Trans. Alg. 3(4) (2007)
29. rsdic - Compressed Rank Select Dictionary, `http://code.google.com/p/rsdic/`
30. Sadakane, K., Navarro, G.: Fully-functional succinct trees. In: SODA 2010, pp. 134–149 (2010)

31. SDSL - Succinct Data Structure Library, `http://www.uni-ulm.de/in/theo/research/sdsl.html`
32. *Succinct* library, `http://github.com/ot/succinct`
33. Sux - Implementing Succinct Data Structures, `http://sux.dsi.unimi.it/`
34. Vigna, S.: Broadword implementation of rank/select queries. In: McGeoch, C.C. (ed.) WEA 2008. LNCS, vol. 5038, pp. 154–168. Springer, Heidelberg (2008)
35. Viola, E.: Bit-probe lower bounds for succinct data structures. SIAM J. Comput. 41(6), 1593–1604 (2012)